DMDB 1.2b 18.6.16

Lecture Summary

Table of Contents

1 Introduction 2
2 Entity-Relationship (ER) Model 2
3 Relational Model 4
4 SQL Examples 6
5 Constraints 7
6 Functional Dependencies 8
7 Normal Forms 10
8 Schemas 12
9 Database Systems 13
10 Query Processing – Algorithms 16
11 Query Optimization 20
12 Transaction Processing 25
13 Atomic Commit 28
14 Replication 32
15 Key Value Stores 35
16 Database Security 36
17 Databases in New Hardware 37

Info

There is no claim for completeness. All warranties are disclaimed. Creative Commons Attribution-Noncommercial 3.0 Unported license.

Study Part

Disclaimer: I have been working with MySQL for a number of years. This probably causes me to be a bit light on coverage of SQL and occasionally slipping in a MySQL-specific detail even though this course is taught with PostgreSQL. I would like to apologize for any inconvenience this may cause.

1 Introduction[1]

What is a database?

A database management system (short: DBMS) is a tool that helps develop and run data-intensive applications. It pushes the complexity of dealing with the data to the database rather than to the program. It allows for the database to be shared. A DBMS can come in many forms and shapes and has many applications. Most databases are relational databases yet recently e.g. graph and NoSQL databases are becoming more and more popular. However, relational databases are still somewhat of a foundation for other types of databases.

The advantages of using a DBMS include avoiding redundancy and inconsistency while recovering after system failures. Access is declarative and concurrent accesses are synchronized. It allows for reusing the data while also providing security and privacy.

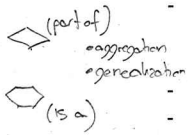
Data Modelling

Conceptual Model	Logical Model (Schema)	Physical Model
It captures the world (domain) to be represented and consists of a collection of entities and how they relate to each other (Entity-Relationship).	It maps the concepts to a concrete logical representation.	It describes the implementation in a concrete hardware architecture.
e.g. ER, UML	e.g. SQLite, COBOL, SQL, XML Infoset, Prolog, ...	

The way data is modelled heavily affects how easy it is to work with the data – choose your model wisely and make sure it works well!

2 Entity-Relationship (ER) Model

The ER model is used at the conceptual stage and, if used correctly, allow for a much cleaner database design. There are a few guidelines and rules of thumb:



- **Avoid redundancy:** the same concept should not be introduced twice (e.g. "assistant" and "professor" – both should inherit from "person"). This leads to the idea of **aggregation and generalization:** there are "is-a" and "part-of" relationships
- **Keep it simple:** simplicity is the key to a good model
- **Attribute vs entity:** a concept is an entity of it has more than one relationship and an attribute if it has only one 1:1 relationship
- **Partitioning of ER models:** since most realistic models are fairly big, it is a good idea to partition it by domain
- **Good vs bad models:** tricks to improve performance are not to be modeled and the fewer entities the better it is

[1] The heading numbers [of level 2] correspond to the chapter/lecture numbers by Prof. Alonso and the subheadings correspond to the different topics outlined in each lecture.

- **C4:** concise, correct, complete, comprehensive

Summary of Notation[2]

This summary contains more elements than we explicitly discussed in class.

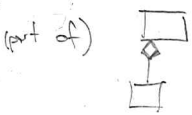
Element	Remarks	Notation
Entity		
Weak Entity	Depends on another entity; always N:1 or 1:1 relationships	
Relationship	Can be recursive (i.e. connect an entity to itself)	
Identifying Relationship	Since weak entities have no keys, an identifying relationship is used	
Attribute		
Key Attribute		
Multivalued Attribute	e.g. a "blue and red" t-shirt	
Composite Attribute		
Derived Attribute	E.g. "age" from "date of birht"	
Total participation of E2 in R ("existence dependency)		E_1 — R — E_2 (weak entity)
Cardinality ratio 1:N for E1:E2 in R		E_1 — 1 R N — E_2
Structural constraint (min,max) on participation of E in R		R — min,max — E
Role		A (single or double) line with a label

Data Modeling with UML

UML (Unified Modeling Language) is a de-facto modeling standard for object-oriented design and is done using class diagrams. While there are a lot of similarities (see table) between ER and UML there are a few important differences (except from the fact of UML having a bigger feature set): methods in UML are associated to classes, keys are not modeled in UML, UML explicitly model aggregation, and UML supports the modeling of instances.

UML term	ER term
Class	Entity
Attribute	Attribute
Association	Relationship
Compositor	Weak entity
Generalization	Generalization

aggregation



[2] Taken from http://web.cse.ohio-state.edu/~gurari/course/cse670/cse670Ch2.xht with minor additions. See also http://creately.com/blog/diagrams/er-diagrams-tutorial/

3 Relational Model RM

Terms

A **relation** R is defined on a **domain** D_i or a Cartesian product thereof, e.g. $R \subseteq D_1 \times \ldots \times D_n$. A relation consists of **tuples** $t \in R$. A **schema** associates labels to domains. The relation variable is also the **table name** of the **table** (relation). A table consists of **attributes** (columns) which are unordered just like the tuples (rows) which are unordered, too.

An example schema is: AddrBook: {[Name: string, Address: string, Tel#:integer]} with an example tuple t = ("Donald Kossmann", "Universitätstrasse", 6).

The **instance** is the state of the database. A **key** is a minimal set of attributes that identify each tuple uniquely. A **primary key** (which is underlined in the schema) is one key used primarily for references.

The difference between a schema and an instance is the following: the schema describes the general associations of attributes to domains ("the model") and an instantiation is just an instance of the schema. Constraints which hold for a particular instance may not be true for the schema.

Basic rules to transform ER to RM

1. Entities become relations
2. Relationships become relations
3. Merge relations with the same key
4. Generalization
5. Weak entities

To name the attributes of relations, proceed as follows: if the ER specifies roles, use the names of the roles, otherwise use the names of the key attributes in the entities (and if ambiguous, come up with new names).

Relational Algebra

Atoms (basic expressions) are either a relation in the database or a constant relation.

	Meaning	Operators	Remarks/Example
σ	Selection	$\sigma_p(E_1)$	Selects tuples from E_1 satisfying predicate p.
π	Projection	$\pi_S(E_1)$	Selects an attribute S from E_1
$\times$	Cartesian product	$E_1 \times E_2$	The Cartesian product of E_1 and E_2 produces $(n+m)$-tuples. This produces a huge result set.
$\bowtie$	(Natural) Join	$E_1 \bowtie E_2$	For two relations R, S with common attributes, a tuple of a (natural) join consists of the attributes $R - S, R \cap S, S - R$. *It matches tuples.*
$\bowtie_\theta$	Theta-join	$E_1 \bowtie_\theta E_2, \theta \in \{<, >, =, \geq, \leq, \ldots\}$	The relations must not have common attributes. For the relation =, it is called an equi-join.
ρ	Rename	$\rho_V(E_1), \rho_{A \leftarrow B}(E_1)$ (el. name, attr. name)	Useful for self-joins and recursive relationships.
$-$	Set minus	$E_1 - E_2$	(same attr)

$R \bowtie_\theta S = \sigma_\theta(R \times S)$

÷	Relation division		Definition: $t \in R \div S$ iff $\forall ts \in S \exists tr \in R$ such that $tr.S = ts.S \wedge tr.(R-S) = t$[3]
∪	Union	$E_1 \cup E_2$	
∩	Intersection		Only works if both relations have the same schema.
⋉	Semi-join (left)	$E_1 \ltimes E_2$	Tuples from the left table matching tuples on the right table.
⋊	Semi-join (right)	$E_1 \rtimes E_2$	Tuples from right table matching tuples on the left table.
⟗	Full outer join	$E_1 ⟗ E_2$	Combination of the left and right outer joins.
⟕	Left outer join	$E_1 ⟕ E_2$	This is a natural join plus unmatched tuples from the left table
⟖	Right outer join	$E_1 ⟖ E_2$	This is a natural join plus unmatched tuples from the right table

Relational Calculus

Queries have the form $\{t \mid P(t)\}$ where t is a variable and $P(t)$ a predicate. There are two variants of relational calculus: tuple relation calculus and domain relation calculus which differ in what the variables iterate over.

Tuple Relational Calculus

Atoms	Formulas
- $s \mid R$: s is a tuple variable, R is a name of a relation - $s.A \phi t.B$ or $s.A \phi c$: s, t tuple variables, A, B attribute names, ϕ a comparison (e.g. $=, \neq, \leq, \ldots$), c is a constant	- All atoms are legal formulas - If P is a formula, then $\neg P$ and (P) are also formulas - If P_1 and P_2 are formulas, then $P_1 \wedge P_2, P_1 \vee P_2, (P_1 \Rightarrow P_2)$ are formulas - If $P(t)$ is a formula with a free variable t, then $\forall t \in R(P(t)), \exists t \in R(P(t))$ are formulas

As a safety measure, formulas are restricted to queries with finite answers (this is a semantic property). **Safety** is defined as follows: the result must be a subset of the domain of the formula whereas said domain consist of all constants, and domains of relations used in the formula.

Domain Relational Calculus

An expression has the form $\{[v_1, v_2, \ldots, v_n] \mid P(v_1, v_2, \ldots, v_n)\}$ where each v_i is either a domain variable or a constant, and P is a formula. Safety is defined in the same way as for tuple relational calculus

Codd's Theorem

The following three languages are equivalent:

1. relational algebra,
2. tuple relational calculus (safe expressions only)
3. domain relational calculus (safe expressions only)

Impact of Codd's theorem:

- SQL is based on the relational calculus
- SQL implementation is based on relational algebra
- Codd's theorem shows that SQL implementation is correct and complete.

[3] See also slide 37

4 SQL Examples

SQL is a Data Definition Language (DDL), Data Manipulation Language (DML), and Query Language.

Relational Division

Since SQL does not support relation division directly, tricks have to be used. Example:

$$\{s \mid s \in \text{Student} \wedge \forall l \in \text{Lecture}\,(l.\text{CP} = 4 \Rightarrow \exists a \in \text{attends}\,(a.\text{Nr} = l.\text{Nr} \wedge a.\text{Legi} = s.\text{Legi}))\}$$

First, $\forall, \Rightarrow$ have to be eliminated:

$$\forall t \in R\,(P(t)) = \neg(\exists t \in R(\neg P(t)))$$
$$R \Rightarrow T = \neg R \vee T$$

This results in

$$\{s \mid s \in \text{Student} \wedge \neg(\exists l \in \text{Lecture}\, \neg\,(\neg(l.\text{CP} = 4) \vee \exists a \in \text{attends}\,(a.\text{Nr} = l.\text{Nr} \wedge a.\text{Legi} = s.\text{Legi}))))\}$$

And after applying DeMorgan rules:

$$\{s \mid s \in \text{Student} \wedge \neg\,(\exists l \in \text{Lecture}\, \neg(l.\text{CP} = 4) \wedge \exists a \in \text{attends}\,(a.\text{Nr} = l.\text{Nr} \wedge a.\text{Legi} = s.\text{Legi}))))\}$$

```
SELECT * FROM Student s
WHERE NOT EXISTS
(SELECT * FROM Lecture l
WHERE l.CP = 4 AND NOT EXISTS
(SELECT * FROM attends a
WHERE a.Nr = l.Nr AND a.Legi=s.Legi));
```

```
SELECT a.Legi FROM attends a
GROUP BY a.Legi
HAVING COUNT(*) = (SELECT COUNT(*) FROM Lecture);
```

NULL

NULL is a **special state** (and not a value!) in SQL which is used when a value is unknown. However, NULLs are treated differently depending on the action used and are also implementation-dependent. NULL values create an open world assumption which contradicts the closed world assumption of the relational model.

When performing arithmetic operations, the NULL value is propagated. For Boolean operations it is considered to be a new Boolean value called "unknown". Whenever a comparison involves NULL, it evaluates to unknown. For WHERE clauses, false and unknown are treated alike. In a GROUP BY statement (provided there are NULL values) there is a group for NULL.

Recursion

While recursion is a natural operation in programming, SQL is not designed with recursion in mind. There exist, however, a few proprietary implementations and it is complex. This is where graph databases come into play.

Snapshot semantics

To ensure deterministic execution of updates (UPDATE, DELETE), a **two-phase process** is used: in the first phase, all by the updates affected tuples are marked and in phase 2 these marked tuples are updated.

Views

Views enable **logical data independence**. This can be useful for privacy/access control, to simplify queries, and for sub/super types to be viewed. A view is similar to a table and is created by storing a query. Contrary to tables, no rows can be inserted and a view can only be updated iff: it involves only one base relation, it involves the key of that base relation, and the view does **not** involve aggregates, group by, or duplicate elimination. As a consequence, updatable views in SQL are a subset of theoretically updatable views.

5 Constraints

The schema defines the domain of the data stored and its concepts which are entities and relations. Types determine the form and space reserved for the attribute values and also influence data processing. To control and ensure **data correctness**, constraints are used. These work similar to pre- and postconditions and control the content of the data and its consistency as part of the schema. This prevents trouble later on. **Transactions** are a tool to manage concurrency control and recovery in case of failures.

Possible problems:

- Inserting tuples without a key
- Adding references to non-existing tuples
- Nonsensical values for attributes
- Conflicting tuples

Examples of constraints include keys, multiplicities of relationships, attribute domains, subset relationship for generalization, and referential integrity (foreign keys). These can either be static (a constraint any instance of a database has to meet) or dynamic (a constraint on a state transition of the database).

Constraints should be implemented on database and application level: the app allows for meaningful error messages whereas the database is a central point, potentially more long-living, allows for DB-level query optimizations, and acts as a safety net.

Referential Integrity Constraints

A **foreign key** refers to a tuple from a different relation. **Referential integrity** is defined as follows: for every foreign key one of these two conditions must hold: the value of the foreign key is NULL *or* the referenced tuple must exist. To maintain referential integrity on update or delete, different actions can be taken:[4]

CASCADE	RESTRICT	NO ACTION	SET DEFAULT/NULL
CASCADE will propagate the change when the parent changes.	RESTRICT causes you cannot delete a given parent row if a child row exists that references the value for that parent row.	Similar to RESTRICT. When a change is executed on the referenced table, the DBMS verifies at the end of the statement execution that none of the referential relationships are violated.	Sets the column value to NULL / default value when you delete the parent table row.

Constraints can be enforced on domains such as integers or enums (enumerations). Constraints can also be enforced across attributes (e.g. for time spans).

[4] http://stackoverflow.com/a/23191693

Triggers

The database can trigger certain actions given a condition when an event occurs (**ECA rule**). Oftentimes integrity constraints are implemented as system triggers. Note however, triggers can be dangerous (consider the marriage example[5])

6 Functional Dependencies

The quality of a schema depends on its amount of redundancy, integrity constrains, and formal properties (which lead to normal forms). It can be improved with synthesis and decomposition algorithms.

Redundancy is bad as it wastes storage space, and requires additional work and code to keep multiple copies of data consistent. However, it is also important as it allows for improved locality, fault tolerance, and availability. Additionally, space (storage and memory) is no longer that big of a problem as it used to be. Note that views imply redundancy.

Anomalies are problems that occur in terms of additional work that needs to be done when changes occur. If the additional work is not done, the database might be inconsistent. This can occur when inserting, updating, and deleting records. As storage becomes cheaper, a strategy is to never throw anything away (**multi-version databases**): instead of deleting, a flag is set. And instead of updating a new version of the tuple is created. Insert anomalies may still exist (multiple NULL values) but these are not inconsistencies.

Functional Dependencies

$$\text{Schema: } \mathcal{R} = \{A: D_A, B: D_B, C: D_C, D: D_D\}$$
$$\text{Instance: } R$$
$$\text{Let } \alpha \subseteq \mathcal{R}, \beta \subseteq \mathcal{R}$$
$$\alpha \to \beta \text{ iff } \forall r, s \in R: r.\alpha = s.\alpha \Rightarrow r.\beta = s.\beta$$
$$\text{equivalent: there is a function } f: X\ D_\alpha \to Y\ D_\beta$$

"Given a relation R, a set of attributes X in R is said to functionally determine another set of attributes Y, also in R, (written $X \to Y$) if, and only if, each X value in R is associated with precisely one Y value in R; R is then said to satisfy the functional dependency $X \to Y$. Equivalently, the projection $\pi_{X,Y}R$ is a function, i.e. Y is a function of X. In simple words, if the values for the X attributes are known (say they are x), then the values for the Y attributes corresponding to x can be determined by looking them up in any tuple of R containing x. Customarily X is called the determinant set and Y the dependent set. A functional dependency $FD: X \to Y$ is called trivial if Y is a subset of X. In other words, a dependency $FD: X \to Y$ means that the values of Y are determined by the values of X. Two tuples sharing the same values of X will necessarily have the same values of Y."[6]

As an analogy to functions: if there is a comma on the left side it indicates a Cartesian product, if there is a comma on the right side it indicates there are multiple functions.

Keys

- $\alpha \subseteq \mathcal{R}$ is a superkey iff $\alpha \to \mathcal{R}$
- $\alpha \to \beta$ is minimal iff $\forall A \in \alpha: \neg((\alpha - \{A\}) \to \beta)$

[5] Slides 21 ff.
[6] https://en.wikipedia.org/wiki/Functional_dependency

- Notation for minimal functional dependencies: $\alpha \dot{\rightarrow} \beta$ (there is a dot superscripted to the arrow)
- $\alpha \subseteq \mathcal{R}$ is a key (or candidate key) iff $\alpha \rightarrow \mathcal{R}$ and it's minimal

Cardinalities define functional dependencies and functional dependencies in turn determine keys. Not all functional dependencies are derived from the cardinality information.

Translating from functional dependencies to a schema

Armstrong Axioms: inference of FDs	Other rules
- Reflexivity: $(\beta \subseteq \alpha) \Rightarrow \alpha \rightarrow \beta$; special case: $\alpha \rightarrow \beta$	- Union: $\alpha \rightarrow \beta \wedge \alpha \rightarrow \gamma \Rightarrow \alpha \rightarrow \beta\gamma$
- Augmentation: $\alpha \rightarrow \beta \Rightarrow \alpha\gamma \rightarrow \beta\gamma$; notation $\alpha\gamma := \alpha \cup \gamma$	- Decomposition: $\alpha \rightarrow \beta\gamma \Rightarrow \alpha \rightarrow \beta \wedge \alpha \rightarrow \gamma$
- Transitivity: $\alpha \rightarrow \beta \wedge \beta \rightarrow \gamma \Rightarrow \alpha \rightarrow \gamma$	- Pseudo-transitivity: $\alpha \rightarrow \beta \wedge \gamma\beta \rightarrow \delta \Rightarrow \alpha\gamma \rightarrow \delta$

These axioms are complete. All possible other rules can be derived from these axioms

The **closure of attributes** takes as input a set of FDs F and α a set of attributes and outputs $\alpha+$ such that $\alpha \rightarrow \alpha+$ which is everything that can be derived from α. This is done by using the reflexivity and transitivity axioms.

Fc is a **minimal basis** of F iff:

1. $Fc \equiv F$: the closure of all attribute sets is the same in Fc and F
2. All function dependencies in F are minimal:
   $\forall A \in \alpha: \left(Fc - (\alpha \rightarrow \beta) \cup \left((\alpha - \{A\}) \rightarrow \beta\right)\right) \not\equiv Fc$
   $\forall B \in \beta: \left(Fc - (\alpha \rightarrow \beta) \cup \left(\alpha \rightarrow (\beta - \{B\})\right)\right) \not\equiv Fc$
3. In Fc, there are no two function dependencies with the same left side; this can be achieved by applying the union rule.

To **compute** the minimal basis:

1. Reduction of left sides of FDs:
   Let $\alpha \rightarrow \beta \in F, A \in \alpha$: if $\beta \subseteq \text{Closure}(F, \alpha - A)$ then replace $\alpha \rightarrow \beta$ with $(\alpha - A) \rightarrow \beta$ in F
2. Reduction of right sides of FDs:
   Let $\alpha \rightarrow \beta \in F, B \in \beta$: if $B \subseteq \text{Closure}\left(F - \left((\alpha \rightarrow \beta) \cup \left(\alpha \rightarrow (\beta - B)\right)\right), \alpha\right)$ then replace $\alpha \rightarrow \beta$ with $\alpha \rightarrow (\beta - B)$ in F
3. Remove FDs: $\alpha \rightarrow \emptyset$ (clean-up of step 2)
4. Apply union rule to FDs with the same left side.

Bad relations combine several concepts and thus such relations need to be **decomposed** such that each concept is in one relation.

1. Lossless decomposition: $\mathcal{R} = \mathcal{R}_1 \bowtie \mathcal{R}_2 \bowtie \cdots \bowtie \mathcal{R}_n$
2. Preservation of dependencies: $FD(\mathcal{R})+ = \left(FD(\mathcal{R}_1) \cup \ldots \cup FD(\mathcal{R}_n)\right)+$

Lemma:

A decomposition is **lossless** for $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$ where $R_1 := \Pi_{\mathcal{R}_1}(R)$ and $R_2 := \Pi_{\mathcal{R}_2}(R)$ if $(\mathcal{R}_1 \cap \mathcal{R}_2) \rightarrow \mathcal{R}_1 \vee (\mathcal{R}_1 \cap \mathcal{R}_2) \rightarrow \mathcal{R}_2$.

To **preserve dependencies**, let $\mathcal{R}$ be decomposed into $\mathcal{R}_i, i \in \{1, \ldots, n\}$ then $F_{\mathcal{R}} = \cup_{i \in \{1,\ldots,n\}} \mathcal{R}_i$.

7 Normal Forms

Normal forms (help to) enforce quite a few aforementioned aspects: concepts should not be together, finding the key of a relation, superfluous concepts of a relation, and redundancy due to dependencies interacting with each other.

The normal forms build up on each other and require lower normal forms to be satisfied.

1NF First Normal Form

The first normal form states only atomic domains should be used. E.g. For a "parents" table there is a record for every child with the same parents and *not* multiple children in one tuple.

2NF Second Normal Form

$\mathcal{R}$ is in 2NF iff every non-key attribute is minimally dependent on every key. To achieve 2NF often a decomposition into multiple relations is necessary.

When talking about 2NF and ER modelling, violations of 2NF happen when ~~mixing~~ combining an entity with an N:M or 1:N relationship. This can be resolved by separating entity and relationship. *However*, it is no problem to ~~mix~~ combine an entity with a 1:1/N:1 relationship.

3NF Third Normal Form

$\mathcal{R}$ is in 3NF iff for all $\alpha \rightarrow \mathcal{B}$ in $\mathcal{R}$ at least one condition holds:

- $B \in \alpha$ (i.e. $\alpha \rightarrow B$ is trivial)
- B is an attribute of at least one key
- α is a superkey of $\mathcal{R}$

If $\alpha \rightarrow B$ does not fulfill any of these conditions, α is a concept in its own right.

3NF is violated when modelling in ER when several entities are mixed (which may or may not be connected by relationships). This can be resolved by implementing each entity in a separate relation (implement N:M relationships in a separate relation).

3NF implies 2NF.

The **synthesis algorithm** is very useful to achieve 3NF. As input it takes a relation $\mathcal{R}$ and FDs F and produces $\mathcal{R}_1, \ldots, \mathcal{R}_n$ such that: $\mathcal{R}_1, \ldots, \mathcal{R}_n$ is a lossless decomposition of $\mathcal{R}$ with preserved dependencies and all $\mathcal{R}_1, \ldots, \mathcal{R}_n$ are in 3NF:

1. Compute the minimal basis Fc of F
2. For all $\alpha \rightarrow \beta \in Fc$ create $\mathcal{R}_\alpha := \alpha \cup \beta$
3. If exists $\kappa \subseteq \mathcal{R}$ such that κ is a key of $\mathcal{R}$ create: $\mathcal{R}_\kappa := \kappa$ (N.B. $\mathcal{R}_\kappa$ has no non-trivial FDs.)
4. Eliminate $\mathcal{R}_\alpha$ if exists $\mathcal{R}'_\alpha$ such that: $\mathcal{R}_\alpha \subseteq \mathcal{R}'_\alpha$

For an example see slides 15 ff.

BCNF Boyce-Codd Normal Form (BCNF)

$\mathcal{R}$ is in BCNF iff for all $\alpha \rightarrow B$ in $\mathcal{R}$ at least one condition holds:

- $B \in \alpha$ (i.e. $\alpha \to B$ is trivial)
- α is a superkey of $\mathcal{R}$

BCNF implies 3NF. As a result, any schema can be decomposed losslessly into BCNF. However, preservation of dependencies cannot be guaranteed – which trades correctness for efficiency.

The **decomposition algorithm** transforms the input $\mathcal{R}$ into $\mathcal{R}_1, \ldots, \mathcal{R}_n$ such that: $\mathcal{R}_1, \ldots, \mathcal{R}_n$ is a lossless decomposition of $\mathcal{R}$ and $\mathcal{R}_1, \ldots, \mathcal{R}_n$ are in BCNF:

```
result = {R}
while (∃Ri ∈ Z: Ri is not in BCNF))
let α → β be evil in Ri
Ri1 = α ∪ β
Ri2 = Ri - β
result = (result – {Ri}) ∪ {Ri1} ∪ {Ri2}
output(result)
```

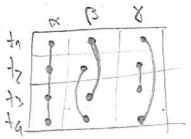
For an example see slides 26 ff.

Multi-value dependencies

"In contrast to the functional dependency, the multivalued dependency requires that certain tuples be present in a relation."[7] MVDs can result in anomalies and redundancy. Furthermore, they are not symmetric.

$\alpha \to\to \beta$ iff $\forall t_1, t_2 \in R: t_1.\alpha = t_2.\alpha \Rightarrow \exists t_3, t_4 \in R:$

$$t_3.\alpha = t_4.\alpha = t_1.\alpha = t_2.\alpha$$
$$t_3.\beta = t_1.\beta, t_4.\beta = t_2.\beta$$
$$t_3.\gamma = t_2.\gamma, t_4.\gamma = t_1.\gamma$$



A decomposition is **lossless** for $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$ where $R_1 := \prod_{\mathcal{R}_1}(R)$ and $R_2 := \prod_{\mathcal{R}_2}(R)$ if $(\mathcal{R}_1 \cap \mathcal{R}_2) \to\to \mathcal{R}_1 \vee (\mathcal{R}_1 \cap \mathcal{R}_2) \to\to \mathcal{R}_2$.

Laws of MVDs:

- Trivial MVDs: $\alpha \to\to \mathcal{R}$ iff $\beta \subseteq \alpha$ or $\beta = \mathcal{R} - \alpha$
- Promotion: $\alpha \to \beta \Rightarrow \alpha \to\to \beta$ (*does not always hold*)
- ~~Reflexivity: $(\beta \subseteq \alpha) \Rightarrow \alpha \to\to \beta$~~
- ~~Augmentation: $\alpha \to\to \beta \Rightarrow \alpha\gamma \to\to \beta\gamma$~~
- ~~Transitivity: $\alpha \to\to \beta \wedge \beta \to\to \gamma \Rightarrow \alpha \to\to \gamma$~~
- Complement: $\alpha \to\to \beta \Rightarrow \alpha \to\to \mathcal{R} - \beta - \alpha$
- Multi-value augmentation: $\alpha \to\to \beta \wedge (\delta \subseteq \gamma) \Rightarrow \alpha\gamma \to\to \beta\delta$
- Multi-value transitivity: $\alpha \to\to \beta \wedge \beta \to\to \gamma \Rightarrow \alpha \to \gamma$
- Generalization (promotion): $\alpha \to \beta \Rightarrow \alpha \to\to \beta$
- Coalesce: $\alpha \to\to \beta \wedge (\gamma \subseteq \beta) \wedge (\delta \cap \beta = \emptyset) \wedge \delta \to \gamma \Rightarrow \alpha \to \gamma$
- Multi-value union: $\alpha \to\to \beta \wedge \alpha \to\to \gamma \Rightarrow \alpha \to\to \beta\gamma$
- Intersection: $\alpha \to\to \beta \wedge \alpha \to\to \gamma \Rightarrow \alpha \to\to (\beta \cap \gamma)$
- Minus: $\alpha \to\to \beta \wedge \alpha \to\to \gamma \Rightarrow \alpha \to (\beta - \gamma) \wedge \alpha \to (\gamma - \beta)$
- **NOT** $\alpha \to\to \beta\gamma \Rightarrow \alpha \to\to \beta \wedge \alpha \to\to \gamma$ (just as an example of not all FD rules holding for MVD)

[7] https://en.wikipedia.org/wiki/Multivalued_dependency

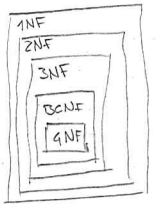
Fourth Normal Form

$\mathcal{R}$ is in 4NF iff for all $\alpha \to\to \beta$ at least one condition holds: $\alpha \to\to \beta$ is trivial or α is a superkey of $\mathcal{R}$. 4NF implies BCNF.

The **decomposition algorithm** transforms the input $\mathcal{R}$ into $\mathcal{R}_1, \ldots, \mathcal{R}_n$ such that: $\mathcal{R}_1, \ldots, \mathcal{R}_n$ are in 4NF:

```
result = {R}
while (∃Ri ∈ Z: Ri is not in 4NF))
let α →→ β be evil in Ri
Ri1 = α ∪ β
Ri2 = Ri - β
result = (result – {Ri}) ∪ {Ri1} ∪ {Ri2}
output(result)
```

almost same



For an example see slide 50.

Summary

Lossless decomposition is ensured up to 4NF, preserved dependencies only up to 3NF.

8 Schemas

A schema captures: the concepts represented including their attributes combined with constraints and dependencies over all attributes.

OLTP: TPC-C

On-line transaction processing (OLTP) refers to workloads with updates, often online (i.e. time requirements), and a high volume of, typically, small transactions (queries return few records and updates affect few records). Integrity is important and OLTP workloads are commonly run over 3NF schemas and many tables. Examples include, banking, online shopping, sales etc.

OLAP: TPC-H

On-line analytical processing (OLAP) refers to workloads with heavy, complex queries (retrieving large number of record, often involving aggregation) and where data is updated rarely and if so in batches. OLAP workloads commonly use de-normalized schema with approaches such as star or snowflake, or data cubes (data mining). Example include marketing analysis, reporting, data analysis etc.

Star schemas

- face t
- dimension t

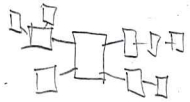
Star schemas are used in OLAP and data mining. It consists of a face table and multiple dimension tables. This schema is used when the design is centered around a very large collection of facts.

Normalization

Normalized tries to avoid redundancy and anomalies yet de-normalized schemas still work (and are used in practice). This has several reasons. OLAP databases are usually not populated in small transaction/by hand but in **batches** which is when constraints and anomalies are controlled. As such this responsibility is shifted away from the schema to data loading application. This makes sense since the data loading application can perform many transformations over the original data (cf. data cubes).

Snowflake

While in a star schema both dimension and fact tables are not normalized, the snowflake schema uses normalized dimension tables (not necessarily all of them). **Normalization** is applied to low cardinality attributes to remove redundancy.



TCP-DS

Omitted.

Modern trends

- Nowadays databases tend to make much more use of memory: the entire databases or the entire working set is main **memory** which removes the need for I/O when processing queries.
- On a physical level, columns are stored rather than rows. → better optimization (same data)
- Combining OLAP and OLTP into one system.
- For heavily-specialize applications, the data is de-normalized (e.g. machine learning).

Data cubes[8]

Data cubes are used for analysis and reporting. They include pre-aggregated data across several dimensions and granularities. Supported operations include **slicing** (selecting over one dimension), **dicing** (selection over the dimensions of the original cube), **drill down** (group-by at finger granularities), and **roll up** (aggregation along a dimension).

9 Database Systems

Reasons to use a database	Tasks a database performs
- Avoid redundancy and inconsistency (query processor, transaction manager, catalog) - Rich/declarative access to data (query processor) - Synchronize concurrent data access (transaction manager) - Recovery after system failures (transaction manager, storage layer) - Security and privacy (server, catalog)	- In takes SQL statement and outputs tuples - It does so by translating SQL statements into get/put requirements for the backend storage and then extracts, processes, and transforms these tuples from blocks - Performs tons of optimizations - Provides security, durability, concurrency control, and further tools

Memory hierarchies

Processing data in a database has always been limited by the problem of **bringing data to the CPU** since there is a lot of data which is always in use and there are lots of queries. While this problem has always been present, its reasons changed over the years – from disks, to main memory and cache hierarchies, and nowadays to the network. This shift also leads to a problem: many current databases still operate under the assumption of data being stored on disk whereas today the data is assumed to be in **main memory**. Another modern problem are multicores which require some sort of synchronization (which is slow). This is especially apparent with cache hierarchies and NUMA[9]. These **cache hierarchies** require very careful data placement and operations are only performed when they make sense since random access to DRAM is very expensive and **sequential access** is preferable.

[8] See PivotTables in Microsoft Excel
[9] Non-uniform memory access:

The **storage manager** therefor has a number of tasks:

Control access to external storage	Management of files and blocks	Buffer management
- Implement the hierarchy for these systems (SSD, disk, tape, ...) - Optimize the heterogeneity of storage - Use OS-level caching to outsmart the file system - Write-ahead logic for redo/undo recovery	- Keep track of files associated with the DB (catalog) - Group set of blocks into pages (granular access)	- Segmentation of buffer pool - Clever replacement policy - Pin pages

Above considerations for storage can also be made for main memory which behaves not altogether different; sequential access is very efficient and random access pollutes the caches and generates faults – and is not predictable.

Storing Data

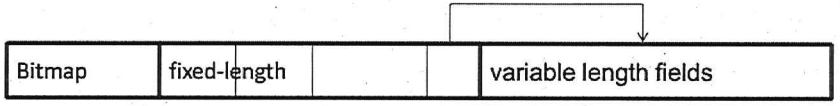
The following table provides a coarse overview of how data is stored:

The **data manager** maps records to pages, the **buffer manager** maps pages in memory to block on disk, and the **catalog** knows where things are.

Records	Tuples
Pages	Collections of tuples of the same table
Blocks	Parts or collection of pages
Tablespaces	Space for a database on a disk
Files	File on disk

The exact notation varies from one database system to another. For example, in Oracle, a **data block** is a page, an **extent** is a set of blocks use for the same purpose (table, index, ...), and a **segment** is a collection of extents stored in the same tablespace.

The **structure of a record** consists of several different parts: **fixed length fields** (e.g. int, data, char) which can be accessed directly, **variable length fields** (e.g. varchar) which store the (length, pointer)-tuple as part of a fixed length field and the payload information in a variable-length field (and thus resulting in a two-step access: 1. retrieve pointer, 2. chase pointer), and **NULL** values in a bitmap (a value of 1 indicates the field is NULL).



An advantage of fixed-length fields is finding the i^{th} field does not require a scan of the record. The information about field types is the same for all records in a file and is stored in **system catalogs**.

For variable-length records, there are two different formats (the number of fields is fixed in both): either storing the field count in the front of the record followed by fields delimited by special symbols *or* an array of field offsets. The latter offers direct access to the i^{th} field, allows for efficient storage of NULLS, and has a small directory overhead.

RID The **record identifier** (RID) stores the page and slot number to identify a record in a **page**. Indexes use these record identifiers to reference records. Records can move within a page, which

might be necessary if they grow or shrank, or are deleted. As a consequence, indexes need not be updated. The **header of a page** stores the slot directory and the number of records in this page.

The **slotted page structure for variable-length records** divides a page into different parts: directory, free space, and data spaces. The slot directory contains the number of record entries, the end of free space in the block, and the location and size of each record. Records are stored at the bottom of the page. External tuple pointers point to record pointers: the record identifier is the page ID and the slot number.

A page may become **full** e.g. when a varchar field is updated such that the record grows and there isn't enough free space in the page anymore. The idea to use is called **row chaining**: it keeps a placeholder (TID), moves the record to a different page, and keeps a "forward" (TID) at the "home" page. If the record moves again, the TID at the "home" page is updated. This is expensive in the long run, requires two I/O to access a record. However, it provides flexibility to move records within and across pages and no updating of references to the record (such as indexes) is needed.

Data Manager

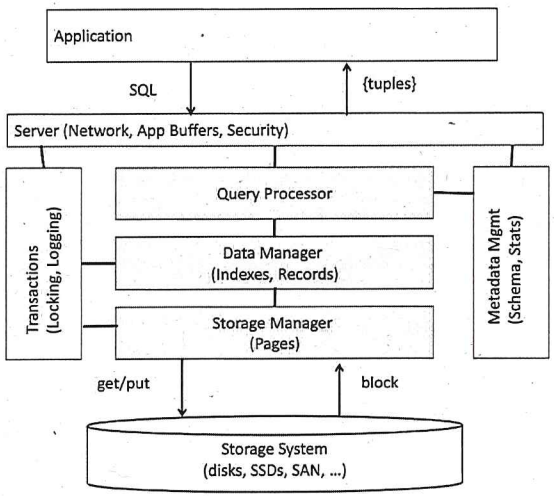
The data manger implements the RIDs by mapping records to pages. Implementing indexes is achieved by using B+ and R trees. An index entry roughly corresponds to a record. Free space is managed by using index-organized tables (IOTs). BLOBs are implemented using variants of position trees.

Essentially, a database is a set of files, whereas a file is to be defined as a variable-sized sequence of blocks (and a block is the transfer unit to disk). A page is fixed-size sequence of blocks and contains records or index entries (in the special case of BLOBs, one record spans multiple pages).

A **table space** explicitly manages files allocated by the database. Each file belongs to a table space, and each table is stored in a table space.

Buffer Management

To minimize disk I/O, pages are kept in memory as long as possible. Thus the questions of replacement policy and transaction manager (i.e. when to write updated pages back to disk) arise. Possible replacement policies are LRU, clock, LRU-k, 2Q (one hot, one cold queue), and MRU. Since the OS also performs paging, there may be a double page fault.



Meta-data Management

All meta-data is stored in tables and internally accessed using SQL. Meta data includes schema, table spaces, histograms, parameters, compiled queries, configuration, user, and workload statistics.

10 Query Processing – Algorithms

There are two main (and one hybrid) approaches for how to process a query: compile it into machine code (better performance) or compile to relational algebra which is then interpreted (easier to debug, better portability).

This chapter, after introducing a few concepts, will deal with the following algorithms for relational algebra:

Table access	Sorting	Joins	Group-by
- Scan	- Two-phase external sorting	- (block) nested loops	- Sorting
- Index scan		- Index nested loops	- Hashing
		- Sort-merge	
		- Hashing	

Basic Concepts

The **query selectivity** is the ratio of number of tuples in the result/output versus the number of tuples in the table/input.

Attribute cardinality refers to how many distinct values an attribute may take. Examples with low cardinality are gender or semester at university while higher cardinality can be found in city and street names.

The **skew** is the probability distribution of the values an attribute takes.

Indexes are a way to avoid a full table scan by building an index over some attributes. Typically, primary and foreign keys are indexed, attributes with high cardinality, and attributes frequently used in joins or as conditions. Yet these indexes are not free, they cost space and maintenance and are not very useful for low selectivity operations. A "bad" index in a student's table would be the gender and a "good" one over the legi/student ID. Indexing can also refer to **partitioning** a table into smaller chunks (for parallel access, caches, increase concurrency, split hot-spots etc.).

Scan versus seek: sequential access is faster than random access – true for disks and also for memory (due to caches). There is a fine tradeoff between doing more reads but sequentially and doing fewer reads but random.

Algorithm: scan

One of the most basic operations – iterating over every tuples and matching it against a predicate. While it may sound expensive, it has a predictable performance which can be desirable.

Algorithm: index scan

If an index is available, many queries can be answered by scanning the index (e.g. range and exist/not exist queries). In a **clustered index**, the leaf nodes in the index contain the data instead of a pointer. While there can be only one such clustered index per table, it allows for direct access to the result.

Sorting in General

Sorting is a very (and necessary – tuples in a table are not sorted) operation, not only as a result but also as an intermediate step for joins, group by, min/max etc. However, it is also expensive, both in CPU and space (not performed in-place with respect to the table).

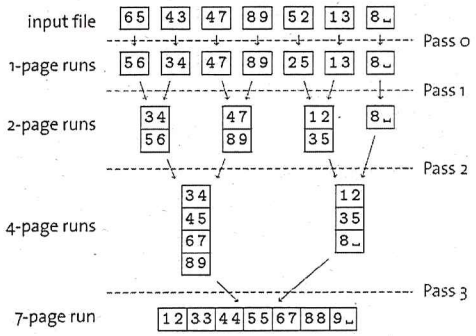
→ Algorithm: two-phase external sorting

The reason for external sorting is on one hand the problem of not being able to fit everything in main memory and on the other hand having multiple queries running at the same time and sharing memory. There are two key parameters: (Exact behavior depends on I/O, CPU, caches etc.)

- N is the number of pages of input
- M is the size of the memory buffer

Phase I: Create Runs	Phase II: Merge Runs	Special Cases
1. Load allocated buffer space with tuples 2. Sort tuples in buffer pool 3. Write sorted tuples (run) to disk 4. Goto Step 1 (create next run) until all tuples processed	Use priority heap to merge tuples from runs	- $M \geq N$: no merge needed - $M < \sqrt{N}$: multiple merge phases necessary

This algorithm may run as a one-pass algorithm (every datum is read once and written once), but when there are many runs, I/O overhead gets very high and the merge step cannot be parallelized. Thus the need for multi-pass versions arises (pictured on the right). This multi-pass version may be executed on different machines or cores.



Asymptotically, the complexity of this algorithm is $O(N \cdot \log N)$. However, this does not account for important CPU and IO complexity which are neglected constants.

Joins in General

Joins are one of the most common operations and can be very expensive[10]. Performance depends on a lot of different factors: actual join algorithm, relative table sizes, number of tables to join, order in which joins are performed, selectivity, predicates in the query, memory hierarchy, and indexes.

→ Algorithm: nested loop join

A nested loop join consists of two nested scans and has complexity $O(|R| \cdot |S|) = O(N^2)$. While this is expensive, it still used as it makes sense if e.g. S is sorted and the join is on an index attribute. An optimization is to use a **block nested loop join** which retrieves several tuples from R in one batch, hashes them and compares them with S – this results in fewer scans of S.

NLJ

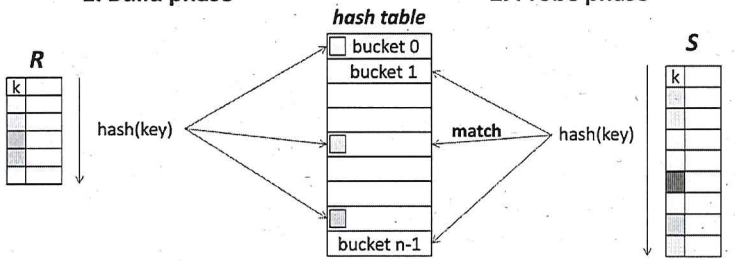
While there are tuples in R
- Get a tuple from R
- Compare with all tuples in S (scan S for matches)
- Output if match

[10] This is ~~very~~ where materialized views come in handy.

→ Algorithm: canonical hash join (classic)

Complexity $O(|R| + |S|) = O(N)$, easy to parallelize.

→ requires enough space to save smaller relation

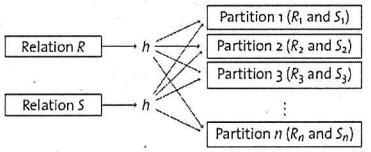


GHJ → Algorithm: grace hash join

```
1 Function: hash_join (R, S, α = β)
2 foreach record r ∈ R do
3     append r to partition R_h(r.α)
4 foreach record s ∈ S do
5     append s to partition S_h(s.β)
6 foreach partition i ∈ 1, ..., n do
7     build hash table H for R_i, using hash function h';
8     foreach block in S_i do
9         foreach record s in current S_i-block do
10            probe H and append matching tuples to result ;
```

h(r.α) → hash function [1...n]

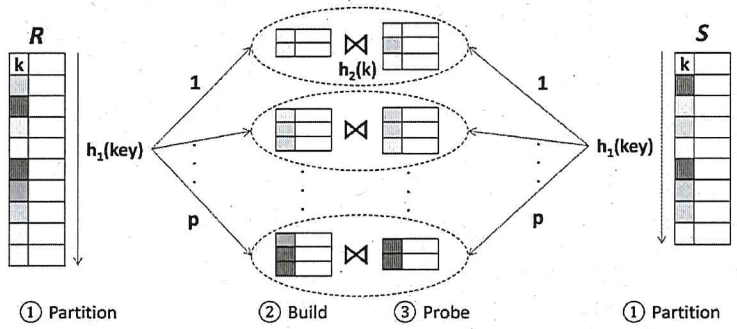
hybrid



$R_i \bowtie S_j = \varnothing$ for all $i \neq j$

→ Algorithm: partitioned hash join

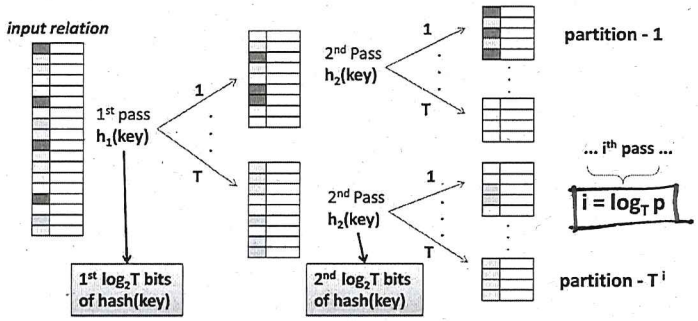
- **Idea:** Partition input into disjoint chunks of cache size
  - No more cache misses during the join



Problem: p can be too large! → p > #TLB-entries → TLB misses; p > #Cache-entries → Cache thrashing

→ Algorithm: Multi-Pass Radix Partitioning

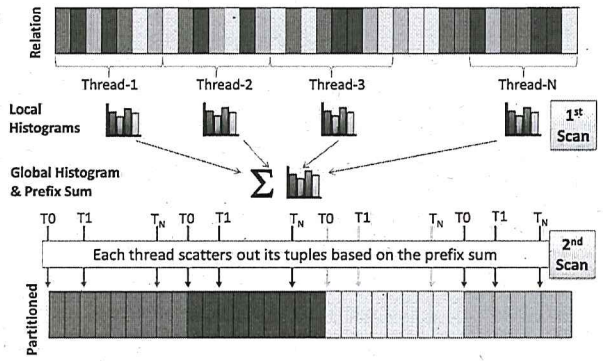
- **Problem:** Hardware limits fan-out, i.e. **T = #TLB-entries** (typically ~~64-512~~)
- **Solution:** Do the partitioning in multiple passes!



- **TLB & Cache** efficiency compensates multiple read/write passes

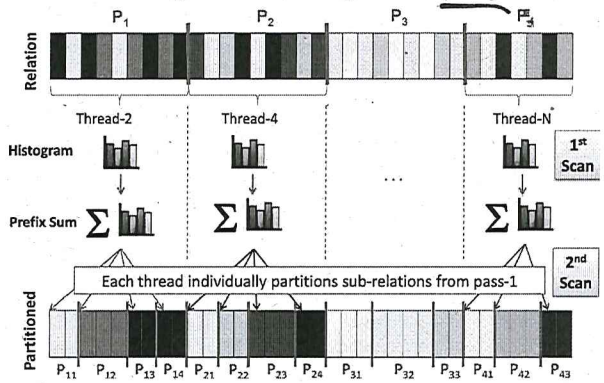
Parallel Radix Join

Parallelizing the Partitioning: *Pass - 1*



Parallel Radix Join

Parallelizing the Partitioning: *Pass - (2 .. i)*



Sorting vs Hashing

Both techniques can be used for joins and group-by and they have the same asymptotic complexity ($O(N \cdot \log N)$, in both, IO and CPU. Hashing has the lower constants for CPU complexity and IO behavior is nearly identical.

Sorting needs merging and hashing needs partitioning. Merging is done *afterward* while partitioning is done *before*. Partitioning depends on good statistics to get it right.

Sorting is more robust, hashing is better in the average case.

Group-by

There are several options:

- Hash on the group-by attribute, aggregate on hash collisions
- Sort on the group-by attribute, then aggregate the sorted ranges
- Choice depend on several factors (like existence of an index)

11 Query Optimization

A query is translated into a plan which is a tree of operators. A plan can be reorganized in many ways which are all equivalent[11]. The optimizer tries to pick the best plan.

Execution Models

The **iterator model** executes the tree using a pipeline (a dataflow graph with records as unit of exchange). Each operator is implemented independently with a generic interface (open(), next(), close()). Each operator is implemented by an iterator. Typical dataflow operator include:

- Union: read both sides, issue all tuples (same schema)
- Union without duplicates: union + remove duplicates (grab one tuple, check against all others, issue if no matches)
- Select (read tuple; while tuple doesn't meet predicate: read tuple; return tuple)
- Projection (read tuple; ~~while more tuples~~: output specified attributes, read tuple)
- Join: trivial nested loop join
- Join with an index on inner relation (for all tuples in r: fetch matching tuple in s <= another operator, output joined tuples if result)

Principle: data flows bottom up in a plan (i.e. operator tree) –control flows top down in a plan

Advantages	Disadvantages
- Generic interface for all operators: great information hiding - Easy to implement iterators (clear what to do in any phase) - Supports buffer management strategies - No overheads in terms of main memory - Supports pipelining: great if only subset of results consumed - Supports parallelism and distribution: add special iterators	- High overhead of method calls - Poor instruction cache locality

Nowadays there are also other models available (e.g., vectorized: block instead of tuple, very fast in column stores) and there are several options to address the limitations of iterators: adaptive execution, non-blocking operators, streaming/partial results, pull vs push, query compilation techniques.

[11] This is one of the main purposes of relational algebra.

Choosing Operators

Selection Operation

File scan	Search algorithms that locate and retrieve records that fulfill a selection condition.
A1 (linear search)	Scan each file block and test all records to see whether they satisfy the selection condition. - Cost estimate (number of disk blocks scanned) = b_r which denotes number of blocks containing records from relation r - If selection is on a key attribute, cost = $(b_{r/2})$; stop on finding record - Linear search can be applied regardless of: selection condition or ordering of records in the file, or availability of indices
A2 (binary search)	Applicable if selection is an equality comparison on the attribute on which file is ordered. - Assume that the blocks of a relation are stored contiguously - Cost estimate (number of disk blocks to be scanned: ▪ $\lceil \log_2 b_r \rceil$ – cost of locating the first tuple by a binary search on the blocks ▪ Plus the number of blocks containing records that satisfy selection condition

Selections Using Indices

Index scan: search algorithms that use an index; selection condition must be on search-key of index.

A3 (primary index on candidate key, equality)	Retrieve a single record that satisfies the corresponding equality condition; cost: $HT_i + 1$
A4 (primary index on non-key, equality)	Retrieve multiple records. Records will be on consecutive blocks. Cost = HT_i + number of blocks containing retrieved records
A5 (equality on search-key of secondary index)	- Retrieve a single record if the search-key *is a candidate key* (cost = $HT_i + 1$) - Retrieve multiple records if search-key *is not a candidate key* (cost = HT_i + number of records retrieved; can be very expensive); each record may be on a different block – one block access for each retrieved record

Selections with Comparisons

Can implement selections of the form $\sigma_{A\leq V}(r)$ or $\sigma_{A\geq V}(r)$
- a linear file scan or binary search
- by using indices in the following ways:

A6 (primary index, comparison)	Relation is sorted on A. For $\sigma_{A\geq V}(r)$ use index to find first tuple $\geq v$ and scan relation sequentially from there For $\sigma_{A\leq V}(r)$ just scan relation sequentially till first tuple $> v$; do not use index
A7 (secondary index, comparison)	For $\sigma_{A\geq V}(r)$ use index to find first index entry $\geq v$ and scan index sequentially from there, to find pointers to records. For $\sigma_{A\leq V}(r)$ just scan leaf pages of index finding pointers to records, till first entry $> v$ In either case, retrieve records that are pointed to: - requires an I/O for each record

- Linear file scan may be cheaper if many records are to be fetched

Complex Selections

Conjunction: $\sigma_{\theta_1} \wedge \sigma_{\theta_2} \wedge \ldots \wedge \sigma\_\theta_n(r)$
Disjunction: $\sigma_{\theta_1} \vee \sigma_{\theta_2} \vee \ldots \vee \sigma\_\theta_n(r)$

A8 (conjunctive selection using one index)	Select a combination of θ_i and algorithms A1 through A7 that results in the least cost for $\sigma_{\theta_i}(r)$ Test other conditions on tuple after fetching it into memory buffer.
A9 (conjunctive selection using multiple-key index)	Use appropriate composite (multiple-key) index if available.
A10 (conjunctive selection by intersection of identifiers).	Requires indices with record pointers. Use corresponding index for each condition, and take intersection of all the obtained sets of record pointers. Then fetch records from file If some conditions do not have appropriate indices, apply test in memory.
A11 (disjunctive selection by union of identifiers).	Applicable if all conditions have available indices (otherwise use linear scan). Use corresponding index for each condition, and take union of all the obtained sets of record pointers. Then fetch records from file.
Negation $\sigma_{\neg\theta}(r)$	Use linear scan on file If very few records satisfy $\neg\theta$, and an index is applicable to θ: find satisfying records using index and fetch from file

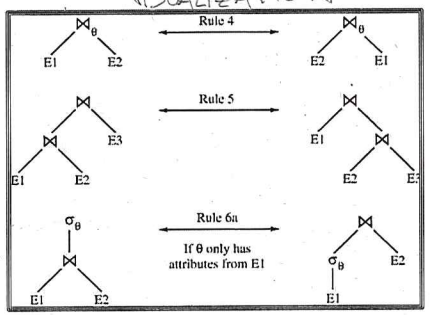
Generating Equivalent Plans

1. Conjunctive selection operations can be deconstructed into a sequence of individual selections.
$$\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$
2. Selection operations are commutative.
$$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$$
3. Only the last in a sequence of projection operations is needed, the others can be omitted.
$$\Pi_{t_1}(\Pi_{t_2}(\ldots(\Pi_{tn}(E))\ldots)) = \Pi_{t_1}(E)$$
4. Selections can be combined with Cartesian products and theta joins.
   a. $\sigma_\theta(E_1 \times E_2) = E_1 \bowtie_\theta E_2$
   b. $\sigma_{\theta 1}(E_1 \bowtie_{\theta 2} E_2) = E_1 \bowtie_{\theta 1 \wedge \theta 2} E_2$
5. Theta-join operations (and natural joins) are commutative.
$$E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$$
6. (a) Natural join operations are associative:
$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$$
(b) Theta joins are associative in the following manner:
$$(E_1 \bowtie_{\theta 1} E_2) \bowtie_{\theta 2 \wedge \theta 3} E_3 = E_1 \bowtie_{\theta 1 \wedge \theta 3} (E_2 \bowtie_{\theta 2} E_3)$$
where θ_2 involves attributes from only E_2 and E_3.
7. The selection operation distributes over the theta join operation under the following two conditions:
(a) When all the attributes in θ_0 involve only the attributes of one of the expressions (E_1) being joined.
$$\sigma_{\theta 0}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta 0}(E_1)) \bowtie_\theta E_2$$
(b) When θ_1 involves only the attributes of E_1 and θ_2 involves only the attributes of E_2.
$$\sigma_{\theta 1 \wedge \theta 2}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta 1}(E_1)) \bowtie_\theta (\sigma_{\theta 2}(E_2))$$
8. The projections operation distributes over the theta join operation as follows:
(a) if L involves only attributes from $L_1 \cup L_2$:
$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = (\Pi_{L_1}(E_1)) \bowtie_\theta (\Pi_{L_2}(E_2))$$
(b) Consider a join $E_1 \bowtie_\theta E_2$.
- Let L_1 and L_2 be sets of attributes from E_1 and E_2, respectively.
- Let L_3 be attributes of E_1 that are involved in join condition θ, but are not in $L_1 \cup L_2$, and
- let L_4 be attributes of E_2 that are involved in join condition θ, but are not in $L_1 \cup L_2$.

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = \Pi_{L_1 \cup L_2}((\Pi_{L_1 \cup L_3}(E_1)) \bowtie_\theta (\Pi_{L_2 \cup L_4}(E_2)))$$

9. The set operations union and intersection are commutative
   - $E_1 \cup E_2 = E_2 \cup E_1$
   - $E_1 \cap E_2 = E_2 \cap E_1$
   - (set difference is not commutative).
10. Set union and intersection are associative.
    $(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$
    $(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$
11. The selection operation distributes over $\cup$, $\cap$ and $-$.
    $\sigma_\theta (E_1 - E_2) = \sigma_\theta (E_1) - \sigma_\theta(E_2)$
    and similarly for $\cup$ and $\cap$ in place of $-$
    Also: $\sigma_\theta (E_1 - E_2) = \sigma_\theta(E_1) - E_2$
    and similarly for $\cap$ in place of $-$, but not for $\cup$
12. The projection operation distributes over union
    $\Pi_L(E_1 \cup E_2) = (\Pi_L(E_1)) \cup (\Pi_L(E_2))$

VISUALIZATION



The earlier selections are processed, the fewer tuples need to be manipulated higher up in the tree (but this may cause losing important ordering of the tuples).

Summarizing query rewriting, the optimizer uses semantically correct rules to transform queries attempting to move constraints between block (because each block is optimized separately) and unnest blocks.

Enumeration of Alternative Plans

The task is to create a query execution plan for a single select-project-join (+ aggregates) block. This is mainly some sort of search through the set of plans while assuming some cost estimation model. In the case of a single block (select, project, aggregation) each available access path is considered and the one with the least estimated cost is chosen.

Query optimization (even ordering of Cartesian products) is NP hard and in general impossible to predict the complexity for a given query. Algorithms used include: dynamic programming (often used), greedy heuristics, randomized algorithms, heuristics relying on hints by programmer, and using a smaller search space.

```
1  Function: find_join_tree_dp (q(R1, ..., Rn))
2  for i = 1 to n do
3      optPlan({Ri}) — access_plans (Ri) ;
4      prune_plans (optPlan({Ri})) ;
5  for i = 2 to n do
6      foreach S ⊆ {R1, ..., Rn} such that |S| = i do
7          optPlan(S) — ∅ ;
8          foreach O ⊂ S do
9              optPlan(S) — optPlan(S) ∪
10                 possible_joins (optPlan(O), optPlan(S \ O));
11         prune_plans (optPlan(S)) ;
12 return optPlan({R1, ..., Rn}) ;
```

Dynamic programming is pictured on the right.

- access_plans: enumerate all ways to scan a table (indexes, ...)
- join_plans: enumerate all ways to join 2 tables (algos, commut.)
- prune_plans: discard sub-plans that are inferior (cost & order)

Plans for Joins

Join plans for R join S	Join plans for three-way joins
- Consider all combinations of access plans - Consider all join algorithms (NL, IdxNL, SMJ, GHJ, ...) - Consider all orders: RxS, SxR - Prune based on cost estimates, interesting orders	- Consider all combinations of joins (assoc., commut.); sometimes even enumerate Cartesian products - Use (pruned) plans of prev. steps as building blocks; consider all combinations - Prune based on cost estimates, interesting orders; interesting orders for the special optimality principle here; gets more complicated in distributed systems

When querying over multiple relations to restrict the search space, e.g. only left-deep join trees can be considered which can then be used to generate all fully pipelined plans. Enumerated using N passes (if N relations joined):

- Pass 1: Find best 1-relation plan for each relation.
- Pass 2: Find best way to join result of each 1-relation plan (as outer) to another relation. (All 2-relation plans.)
- Pass N: Find best way to join result of a (N-1)-relation plan (as outer) to the N'th relation. (All N-relation plans.)

For each subset of relations, retain only the cheapest plan overall, plus the cheapest plan for each interesting order of the tuples.

ORDER BY, GROUP BY, aggregates etc. handled as a final step, using either an interestingly ordered plan or an additional sorting operator. An N-1 way plan is not combined with an additional relation unless there is a join condition between them, unless all predicates in WHERE have been used up (i.e., avoid Cartesian products if possible). In spite of pruning plan space, this approach is still exponential in the number of tables. If all (bushy) trees are to be considered, the algorithm has to be modified only slightly.

Interesting Orders in plans

Consider the expression $(r_1 \bowtie r_2 \bowtie r_3) \bowtie r_4 \bowtie r_5$. An interesting sort order is a particular sort order of tuples that could be useful for a later operation. Generating the result of $r_1 \bowtie r_2 \bowtie r_3$ sorted on the attributes common with r_4 or r_5 may be useful, but generating it sorted on the attributes common only r_1 and r_2 is not useful. Using merge-join to compute $r_1 \bowtie r_2 \bowtie r_3$ may be costlier, but may provide an output sorted in an interesting order.

It is not sufficient to find the best join order for each subset of the set of n given relations; must find the best join order for each subset, for each interesting sort order. This is a simple extension of dynamic programming algorithms. Usually, the number of interesting orders is quite small and doesn't affect time/space complexity significantly.

Heuristic Optimization

Cost-based optimization is expensive, even with dynamic programming. Systems may use heuristics to reduce the number of choices that must be made in a cost-based fashion. Heuristic optimization transforms the query-tree by using a set of rules that typically (but not in all cases) improve execution performance:

- Perform selection early (reduces the number of tuples)
- Perform projection early (reduces the number of attributes)
- Perform most restrictive selection and join operations before other similar operations.
- Some systems use only heuristics; others combine heuristics with partial cost-based optimization

Typical steps in heuristic optimization are:

1. Deconstruct conjunctive selections into a sequence of single selection operations (Equiv. rule 1.).
2. Move selection operations down the query tree for the earliest possible execution (Equiv. rules 2, 7a, 7b, 11).

3. Execute first those selection and join operations that will produce the smallest relations (Equiv. rule 6).
4. Replace Cartesian product operations that are followed by a selection condition by join operations (Equiv. rule 4a).
5. Deconstruct and move as far down the tree as possible lists of projection attributes, creating new projections where needed (Equiv. rules 3, 8a, 8b, 12).
6. Identify those subtrees whose operations can be pipelined, and execute them using pipelining

12 Transaction Processing

A transaction is an ordered sequence of read and writes over the database that either commits or aborts.

A few basic assumptions about transactions:

- Transactions are delimited by a being operation and either an abort or commit operation.
- A transaction must end with a commit or abort but cannot have both.
- After a transaction commits or aborts, no more operations from that transaction are possible.
- Operations within a transaction are totally ordered with respect to each other.
- Communication between transactions only happens through the database by reading and writing.
- Transaction are correct programs i.e. if they operate on a consistent state of the database, they will leave it in a consistent state.

Concurrency control does not imply a time ordering (implementation-dependent). Oftentimes, the ordering imposed by concurrency control is not the same ordering given by time. The goal is to find a way to executes operations in parallel while ensuring a correct (defined in some notion) result. Being correct does *not* necessarily imply respect for the order in time in which *different transactions* are executed. However, it *does* imply to respect the order established *within one* transaction.

A transaction has the following **operations**:

- **Begin of a transaction (BOT)**: often implicit
- **Commit**: the transaction has finished; the database confirms to the client when all changes of the transaction have been made consistent.
- **Abort**: transaction is cancelled; the database rolls back all changes done by the transaction
- **Read**: read a datum
- **Write**: write a datum
- $a <_T b$: a happens before b, with respect to the partial order T

The **ACID**[12] principle is very important when it comes to databases and transactions in particular:

- **Atomicity:** a transaction is executed in its entirety or not at all.
- **Consistency:** a transaction executed in its entirety over a consistent database produces a consistent database.
- **Isolation:** a transaction executed as if it were alone in the system.
- **Durability:** committed changes of a transactions are never lost i.e. they are recoverable.

[12] Yes, a BASE principle (for NoSQL databases) exists.

Transaction management consists of two main (and intereonnected) parts: concurrency control (i.e. enforcing isolation among concurrently running transactions) and recovery (i.e. ensuring durability and atomicity of transactions).

When talking about **histories**, a **conflicting operation** is defined when two operations act on the same item and one of them is a write. A history H is a partially ordered sequence $<_H$ of operations from a set of transactions where:

- If two operations are ordered within a transaction, they are equally ordered in the history.
- If two operations, p and q, conflict then they are ordered with respect to each other.

Abort (reads of T_j are irrelevant)	**Commit**
- $r_i(x), a_j$: conflict if T_j updated x	- $r_i(x), c_j$: no conflict
- $w_i(x), a_j$: conflict if T_j updated x	- $w_i(x), c_j$: no conflict

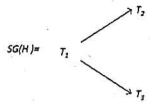
Concurrency Control Theory

A history H is **serial** if, for every two transactions T_i, T_j that appear in H, either all operations from T_i appear before all operations of T_j or vice versa. A serial history with only committed transactions is correct by definition since transactions are isolated and each transaction starts in a consistent state and leaves the database in a consistent state.

The notion of **history equivalence** captures the fact that, in the two histories, committed transactions see the same state (read the same values) and leave the database in the same state. Two histories are equivalent iff:

1. They are over the same transactions and contain the same operations.
2. Conflicting operations of non-aborted transactions are ordered in the same way in both histories.

Following from above facts is that a history is **serializable** iff it is equivalent to a serial history. This is the case for an **acyclic serializability graph**.



Topological Sorting

$H_s^1 = T_1 \mid T_2 \mid T_3$

$H_s^2 = T_1 \mid T_3 \mid T_2$

$H \equiv H_s^1 \equiv H_s^2$

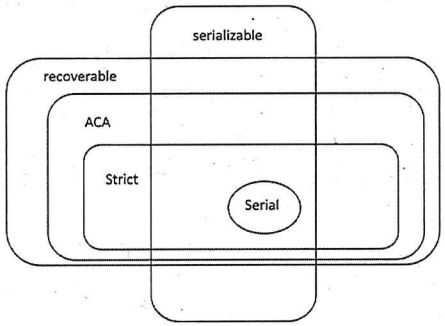
Recovery Theory

There are four types of recovery:

Abort of a single transaction (application, system)	**System crash: lose main memory, keep disk**	**System crash: loss of disks**
- *R1*: undo single transaction	- *R2*: redo committed transactions - *R3*: undo active transactions	- *R4*: read backup of database from tape

UNDO When changes of an aborted transaction are undone, (typically) the image of the value before the modification is restored. To redo an operation, the same process is performed. As a prerequisite databases have to (and do) log transactions by keeping before and after images of the changes.

REDO



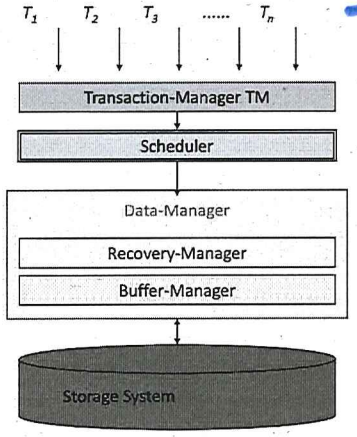
To talk about recovery on histories, a few definitions are necessary:

A transaction T_1 **reads from** another transaction T_2 if T_1 reads a value written by T_2 at a time when T_2 was not aborted.

Name of history type	Property	Implication	Problems if not satisfied
RC **Recoverable (RC)**	If T_i reads from T_j and commits, then $c_j < c_i$.	There is no need to undo a committed transaction. ($c_i < a_j$)	Data read was later removed because another program aborted.
ACA **Avoids cascading aborts (ACA)**	If T_i reads x from T_j, then $c_j < r_i[x]$.	Aborting a transaction does not cause aborting other transactions. ($r_i < a_j$)	Thrashing behavior when transactions keep aborting each other.
ST **Strict (ST)**	If T_i reads from or overwrites a value written by T_j then $c_j < r_i[x]/w_i[x]$ or $a_j < r_i[x]/w_i[x]$.	Undoing a transaction does not undo the changes of other transactions. ($r/w_i < a_j$)	Recovery after a failure becomes very complex (or impossible).

Database Locking

If locking was performed by the operating system, performance would decrease as the OS has to treat all operations as block boxes whereas the database knows the semantics (e.g. read or write). The database uses the so-called **two-phase locking protocol (2PL):** (no ACID)



1. Before access an object, a transaction *must acquire* a lock.
2. A transaction acquires a lock *only once*. Lock upgrades are possible.
3. A transaction is *blocked* if the lock request cannot be granted according to the compatibility matrix.
4. A transaction goes through *two phases*:
   Growth: acquire locks, but never release a lock
   Shrink: release lock, but never acquire a lock
5. At EOT[13] all locks must be *released*.

[13] End of transaction, either commit or abort

Note: an abort of one transaction may cause the abort of another transaction, possibly even leading to a domino effect. This issue leads to an improvement of 2PL: **Strict 2PL**. In Strict 2PL all locks are kept until EOT. This avoids cascading aborts, avoids rollback of committed transactions[14], and couples visibility[15] with recoverability[16].

Detecting **deadlocks** can either happen by using a wait-for graph or by detecting timeouts.

SI **Snapshot Isolation** (→ no serializability)

Snapshot isolation refers to the concept of separating readers from writers by giving readers only a **copy of the database** (to work on) while writers **create a new copy** of the database (which is made visible at commit time).

T When a transaction starts, it receives a timestamp T. All **reads** are carried out as of the database version at T (thus older data needs to be kept) whereas all **writes** are carried out in a separate buffer (and only become visible after commit). When a transaction **commits**, the database checks for conflicts. If an early transaction has a conflicting write and committed after another transaction started, then abort. Snapshot isolation does *not provide serializability.*

Concurrency and Availability	Overhead	Problem: Write Skew
- No read or write is ever blocked - Only commits can be blocked	- Need to keep write-set of a TA only - Very efficient way to implement aborts (do nothing) - Often keeping all versions of an object useful anyway - No deadlocks, but unnecessary rollbacks (could have been avoided)	- Checking integrity constraint also happens in the snapshot - Two concurrent TAs update different objects - Each update okay, but combination not okay

2PL and snapshot isolation have a different notion of time and history:

Definition of History	Histories in 2PL	Histories in SI
- Partial order of all operations - Total order of conflict operations	- Partial order of all operations - Total order of conflict operations - No re-ordering of operations - Only serializable histories (modulo phantoms)	- Total order of all operations (supports versioning of data) - Re-ordering of R-W conflict operations - Abort to deal with W-W conflict operations - Allows non-serializable histories

13 Atomic Commit

Distributed **consensus** is the problem of reaching an agreement among all working processes on the value of a variable. If the system is reliable, this is easy to achieve. Yet in an asynchronous system (where no assumptions about timing can be made and thus it is impossible to distinguish between a failure and a slow system) it is *impossible*. All entirely asynchronous commit protocols may block and no commit protocol can guarantee independent recovery.

[14] As such, Strict 2PL implements ACID properly, in contrast in basic 2PL.
[15] Property of Strict 2PL.
[16] ACID properties A and D.

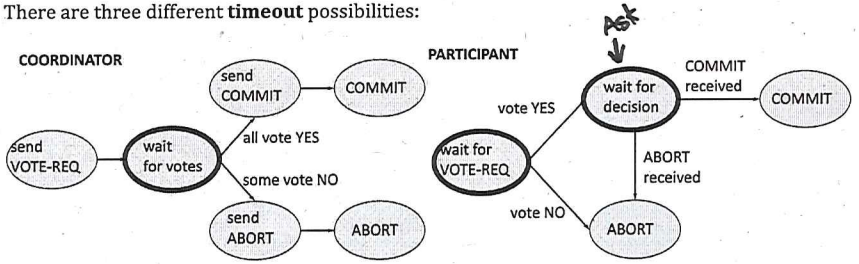
Enforcing atomic commitment consist of five properties:

AC1 All processors that reach a decision reach the same one (agreement, consensus).
AC2 A processor cannot reverse its decision.
AC3 Commit can only be decided if all processors vote YES (no imposed decisions).
AC4 If there are no failures and all processors voted YES, the decision will be to commit (non triviality).
AC5 Consider an execution with normal failures. If all failures are repaired and no more failures occur for sufficiently long, then all processors will eventually reach a decision (liveness).

The Two-Phase Commit Protocol (2PC)

Protocol	Correctness
- Coordinator sends VOTE-REQ to all participants. - Upon receiving a VOTE-REQ, a participant sends a message with YES or NO (if the vote is NO, the participant aborts the transaction and stops). - Coordinator collects all votes: All YES = Commit and send COMMIT to all others. Some NO = Abort and send ABORT to all which voted YES. - A participant receiving COMMIT or ABORT messages from the coordinator ~~decides accordingly~~ and stops.	AC1 Every processor decides what the coordinator decides (if one decides to abort, the coordinator will decide to abort). AC2 Any processor arriving at a decision "stops". AC3 The coordinator will decide to commit if all decide to commit (all vote yes). AC4 If there are no failures and everybody votes yes, the decision will be to commit. AC5 The protocol needs to be extended in case of failures (in case of timeout, a site may need to "ask around")

There are three different **timeout** possibilities:



- If the coordinator times-out waiting for votes: it can decide to abort (nobody has decided anything yet, or if they have, it has been to abort).
- If a participant times-out waiting for VOTE-REQ: it can decide to abort (nobody has decided anything yet, or if they have, it has been to abort).
- If a participant times-out waiting for a decision: it cannot decide anything unilaterally, it must ask (run a Cooperative Termination Protocol). If everybody is in the same situation no decision can be made: all processors will block. This state is called uncertainty period.

When in doubt, ask. If anybody has decided, they will tell say what the decision was:

- There is always at least one processor that has decided or is able to decide (the coordinator has no uncertainty period). Thus, if all failures are repaired, all processors will eventually reach a decision.

- If the coordinator fails after receiving all YES votes but before sending any COMMIT message: all participants are uncertain and will not be able to decide anything until the coordinator recovers. This is the blocking behavior of 2PC (compare with the impossibility result discussed previously).

Processors must know their state to be able to tell others whether they have reached a decision. This state must be persistent.

Persistence is achieved by writing a log record.

This requires flushing the log buffer to disk (I/O). This is done for every state change in the protocol. This is done for every distributed transaction.

And thus this is expensive.

Events for which a log entry is written:

- When sending VOTE-REQ, the coordinator writes a START-2PC log record (to know the coordinator).
- If a participant votes YES, it writes a YES record in the log BEFORE it sends its vote. If it votes NO, then it writes a NO record.
- If the coordinator decides to commit or abort, it writes a COMMIT or ABORT record before sending any message.
- After receiving the coordinator's decision, a participant writes its own decision in the log.

In a linear 2PC the total number of messages is $2n$ (instead of $3n$), but the number of rounds is $2n$ (instead of 3).

Processes in 2PC may **block**: if a process fails and everybody else is uncertain, there is no way to know whether this process has committed or aborted.[17] Yet an uncertain process cannot make a decision is that being uncertain does not mean all other processes are uncertain. Processes may have decided and then failed.

3PC[18]

Two versions of 3PC will be discussed:

- One capable of tolerating only site failures (no communication failures). Blocking occurs only when there is a total failure (every process is down). This version is useful if all participants reside in the same site.
- One capable of tolerating both site and communication failures (based on quorums). But blocking is still possible if no quorum can be formed.

A new rule, the **non-blocking rule (NB rule)** is introduced: No operational process can decide to commit if there are operational processes that are uncertain. This rule guarantees that if anybody is uncertain, nobody can have decided to commit. Thus, when running the cooperative termination protocol, if a process finds out that everybody else is uncertain, they can all safely decide to abort. The consequence of this rule is that the coordinator cannot make a decision by itself as in 2PC. Before making a decision, it must be sure that everybody is out of the uncertainty area. Therefore, the coordinator, must first tell all processes what is going to happen: (request votes, prepare to commit, commit). This implies yet another round of messages.

[17] The coordinator has no uncertainty period. To block the coordinator must fail.
The fact that everybody is uncertain implies everybody voted YES.
[18] Not used in practice

3PC is interesting in that the processes know what will happen before it happens:

- Once the coordinator reaches the "bcast pre-commit", it knows the decision will be to commit.
- Once a participant receives the pre-commit message from the coordinator, it knows that the decision will be to commit.

The extra round of messages is used to spread knowledge across the system. These messages provide information about what is going on at other processes (NB rule).

The NB rule is used when time-outs occur[19]:

- If coordinator times out waiting for votes = ABORT.
- If participant times out waiting for votereq = ABORT.
- If coordinator times out waiting for ACKs = ignore those who did not sent the ACK! (at this stage everybody has agreed to commit).
- If participant times out waiting for precommit = still in the uncertainty period, ask around.
- If participant times out waiting for commit message = not uncertain any more but needs to ask around.

Similarly to 2PC, a process has to remember its previous actions to be able to participate in any decision. This is accomplished by recording every step in the log:

- Coordinator writes "start-3PC" record before doing anything. It writes an "abort" or "commit" record before sending any abort or commit message.
- Participant writes its YES vote to the log before sending it to the coordinator. If it votes NO, it writes it to the log after sending it to the coordinator. When reaching a decision, it writes it in the log (abort or commit).

Processes in 3PC cannot independently recover unless they had already reached a decision or they have not participated at all:

- If the coordinator recovers and finds a "start 3PC" record in its log but no decision record, it needs to ask around to find out what the decision was. If it does not find a "start 3PC", it will find no records of the transaction, then it can decide to abort.
- If a participant has a YES vote in its log but no decision record, it must ask around. If it has not voted, it can decide to abort.

Termination Protocol:

- Elect a new coordinator.
- New coordinator sends a "state req" to all processes. Participants send their state (aborted, committed, uncertain, committable).
- TR1 = If some "aborted" received, then abort.
- TR2 = If some "committed" received, then commit.
- TR3 = If all uncertain, then abort.
- TR4 = If some "committable" but no "committed" received, then send "PRE-COMMIT" to all, wait for ACKs and send commit message.

[19] Remember, however, that there are no communication failures.

Even though TR4 is similar to 3PC, the problem is solved because failures of the participants on the termination protocol can be ignored. At this stage, the coordinator knows that everybody is uncertain, those who have not sent an ACK have failed and cannot have made a decision. Therefore, all remaining can safely decide to commit after going over the precommit and commit phases. The problem is when the new coordinator fails after asking for the state but before sending any precommit message. In this case, it needs have to start all over again.

This protocol does not tolerate communication failures:

- A site decides to vote NO, but its message is lost.
- All vote YES and then a partition occurs. Assume the sides of the partition are A and B and all processes in A are uncertain and all processes in B are committable. When they run the termination protocol, those in A will decide to abort and those in B will decide to commit.
- This can be avoided if quorums are used, that is, no decision can be made without having a quorum of processes who agree (this reintroduces the possibility of blocking, all processes in A will block).

Total failures require special treatment, if after the total failure every process is still uncertain, it is necessary to find out which process was the last one to fail. If the last one to fail is found and is still uncertain, then all can decide to abort. The reason for this are partitions. Everybody votes YES, then all processes in A fail. Processes in B will decide to commit once the coordinator times out waiting for ACKs. Then all processes in B fail. Processes in A recover. They run the termination protocol and they are all uncertain. Following the termination protocol will lead them to abort.

14 Replication

Replication is very common with databases: RAID, mirroring, backups (including standby). There are three main reasons why replication is desirable:

- **Performance**: Location transparency is difficult to achieve in a distributed environment. Local accesses are fast, remote accesses are slow. If everything is local, then all accesses should be fast.
- **Fault tolerance**: Failure resilience is also difficult to achieve. If a site fails, the data it contains becomes unavailable. By keeping several copies of the data at different sites, single site failures should not affect the overall availability.
- **Application type**: Databases have always tried to separate queries form updates to avoid interference. This leads to two different application types OLTP and OLAP, depending on whether they are update or read intensive.

There are two basic parameters to select from when designing a replication strategy: when (synchronous/eager and asynchronous/lazy) and where (primary copy/master and update everywhere/group).

	Synchronous	Asynchronous
Description	Synchronous replication propagates any changes to the data immediately to all existing copies. Moreover, the changes are propagated within the scope of the transaction making the changes. The ACID properties apply to all copy updates.	Asynchronous replication first executes the updating transaction on the local copy. Then the changes are propagated to all other copies. While the propagation takes place, the copies are inconsistent (they have different values). The time the copies are inconsistent is an adjustable parameter which is application dependent.
Pros	- No inconsistencies (identical copies) - Reading the local copy yields the most up to date value - Changes are atomic	- A transaction is always local (good response time)
Cons	- A transaction has to update all sites (longer execution time, worse response time)	- Data inconsistencies - A local read does not always return the most up to date value - Changes to all copies are not guaranteed - Replication is not transparent

	Primary copy	Update everywhere
Descr.	With an update everywhere approach, changes can be initiated at any of the copies. That is, any of the sites which owns a copy can update the value of the data item.	With a primary copy approach, there is only one copy which can be updated (the master), all others (secondary copies) are updated reflecting the changes to the master.
Pros	- No inter-site synchronization is necessary (it takes place at the primary copy) - There is always one site which has all the updates	- Any site can run a transaction - Load is evenly distributed
Cons	- The load at the primary copy can be quite large - Reading the local copy may not yield the most up to date value	- Copies need to be synchronized

Putting everything together, the following matrix results:

	Primary copy	Update everywhere
Synchronous	*Advantages*: - Updates do not need to be coordinated - No inconsistencies *Disadvantages*: - Longest response time - Only useful with few updates - Local copies are can only be read → *too expensive in practice*	*Advantages*: - No inconsistencies - Elegant (symmetrical solution) *Disadvantages*: - Long response times - Updates need to be coordinated → *does not scale in practice*
Asynchronous	*Advantages*: - No coordination necessary - Short response time *Disadvantages*: - Local copies are not up to date - Inconsistencies → *feasible in practice*	*Advantages*: - No centralized coordination - Shortest response times *Disadvantages*: - Inconsistencies - Updates can be lost (reconciliation) → *feasible for some applications*

Replication Protocols

Also when talking about replication, the ACID properties[20] have to be enforced/maintained. Atomicity is guaranteed by using 2PC. The problem is how to make sure the **serialization orders are the same** at all sites, i.e., make sure that all sites do the same things in the same order (otherwise the copies would be inconsistent). To achieve this, replication protocols are used. A **replication protocol** specifies how the different sites must be coordinated in order to provide a concrete set of guarantees. The replication protocols depend on the replication strategy.

Replication is associated with a **cost**, where the fraction of data being replicated is s and the w is the fraction of updates made (i.e. ws is the replication factor), and n is the number of nodes, the overall computing power of the system is:

$$\frac{n}{1 + w \cdot s \cdot (n-1)}$$

Increasing ws does *not* result in a performance gain – only local reads result in performance advantages.

Assuming all sites contain the same data, one could use the **read one – write all** protocol (synchronous, update everywhere):

- Each sites uses 2 Phase Locking.
- Read operations are performed locally.
- Write operations are performed at all sites (using a distributed locking protocol).

This protocol guarantees that every site will behave as if there were only one database. The execution is serializable (correct) and all reads access the latest version. And it also illustrates the main idea behind replication, but it needs to be extended in order to cope with realistic environments:

- Sites fail, which reduces the availability (if a site fails, no copy can be written).
- Sites eventually have to recover (a recently recovered site may not have the latest updates).

Assuming there are no communication failures (only site failures), one could use **write all available copies**:

- READ = read any copy, if time-out, read another copy.
- WRITE = send Write(x) to all copies. If one site rejects the operation, then abort. Otherwise, all sites not responding are "missing writes".
- VALIDATION = To commit a transaction:
  Check that all sites in "missing writes" are still down. If not, then abort the transaction.
  Check that all sites that were available are still available. If some do not respond, then abort.

In practice, communications may fail leading to partitioned networks and inaccessible sites. The available copies algorithm assumes that when a site cannot be reached, it must have failed. To tolerate site and communication failures, **quorum protocols** are used. Quorums are sets of sites formed in such a way so as to be able to determine that it will have a non-empty intersection with other quorums:

[20] Atomicity, consistency, isolation, durability

- Simple majorities (site quorums).
- Weighted majorities (quorum consensus).
- Logical structures (tree, matrix, projective planes quorums).

Desirable properties of quorums:

- Equal effort: all quorums should have about the same size.
- Equal responsibility: all sites participate on the same number of quorums.
- Dynamic reconfiguration: establishing a quorum should be dynamic to account for failures and configuration changes.
- Low communication overhead: minimize the number of messages exchanged.
- Graceful degradation: effort proportional to failures

As an example of weighted quorums serves the **quorum consensus protocol**:

- Each copy has a weight assigned to it.
- The total weight of all copies is N.
- Let RT and WT be read and write thresholds (resp.), such that: $2WT > N$ and $RT + WT > N$
- A read quorum is a set of copies such that their total weight is greater or equal to RT.
- A write quorum is a set of copies such that their total weight is greater or equal to WT.

Furthermore: each copy has a version number.

- READ = contact sites until a read quorum is formed. Then read the copy with the highest version number.
- WRITE = contact sites until a write quorum is formed. Get the version number of the copy with the highest version number (k). Write to all sites in the quorum adding the new version number (k+1).

Recovery is for free, but reading is no longer local and it is not possible to change the system dynamically (the copies must be known in advance).

One of the **main problems** of synchronous + update everywhere is the way replication takes place (one operation at a time increases the response time and, thereby, the conflict profile of the transaction. The message overhead is too high (even if broadcast facilities are available). Additionally, the probability for deadlocks is very high. Synchronous + primary copy faces similar problems.

Asynchronous + update everywhere fixes most of these problems but faces issues when faced with **reconciliation**. This has to be solved almost manually, yet there are strategies:

- Latest update win (newer updates preferred over old ones)
- Site priority (preference to updates from headquarters)
- Largest value (the larger transaction is preferred)
- Identify the changes and try to combine them
- Analyze the transactions and eliminate the non-important ones
- Custom priority schemas

15 Key Value Stores

KVS dictionary / hash

A KVS does **not** have:

- A schema: just a key plus a blob attached to it

- SQL: only get and set
- ACID: eventual consistency, potentially no transactions

blob / blog

Schema	Deployment	Replication strategy
- Key + blob / pointer to blob - Index on key (hashing)	- Horizontal partitioning - Replication of partitions	- Quorums (simple majority) - Asynchronous replication across all copies (eventual consistency)

The Good	The Bad	The Ugly
- Very fast lookups (hashing) - Easy to scale to many machines (simply add more copies/machines) - Useful in many applications: Lookup user profiles Retrieve users web pages and info Scalable and easy to get going (no schema)	- No easy support for queries beyond point queries (ranges, joins; queries on attributes (there are no attributes)) - Often, data is inconsistent: application-dependent correctness	- Pushes complexity and responsibility to the application – price will be paid, the question is where - Some operations are very costly to implement (range queries require to read all copies; joins out of the question) - Works well as cache or specialized system - Not a replacement for a full blown database

Most serious systems are converging back towards a full blown transactional engine with a schema and SQL support. Specialized systems in used as accelerators (memcached) or concrete application stages (profile look up).

16 Database Security

"Data Security is the science and study of methods of protecting data (...) from unauthorized disclosure and modification."[21] Thus data security = confidentiality + integrity.

There are three main security tasks in databases:

AAA
- authentication
- authorization
- auditing

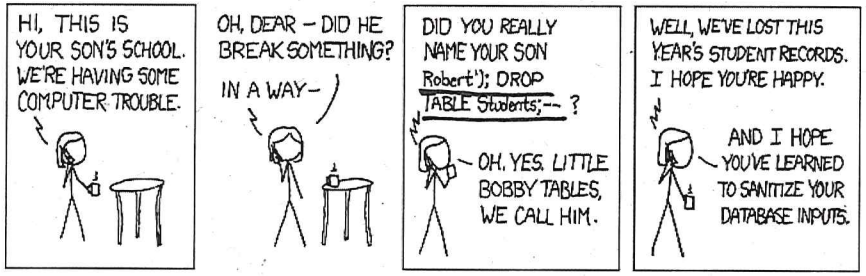
- Authentication: verifying the id of a user
- Authorization: checking the access privileges
- Auditing: looking for violations (in the past)

Privacy (Confidentiality):

- Do not allow access to private information
- But support statistical analyses over private information
- Avoid inference attacks (ask other things to deduce)
- Q does not leak info for secret S if: $P(S|Q) = P(S)$

SQL injection can be prevented by using prepared statements and checking input validity.

[21] Dorothy Denning, 1982


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH, YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

From https://xkcd.com/327/, see also https://www.explainxkcd.com/wiki/index.php/Little_Bobby_Tables

Access in control in SQL works by granting specific privileges over tables or views. This is very coarse. Users can be assigned to groups. There is a capability GRANT which allows users to grant other uses privileges.

Please also see the respective slide deck.

17 Databases in New Hardware

Omitted.

Sources

Unless otherwise noted: Lecture slides by Gustavo Alonso available on the course website accompanying the course 252-0063-00L taught in the spring semester 2016 at ETH Zürich. Simple definitions might be from Wikipedia.